

Automatic Measurement of Hardware Parameters for Embedded Processors^{*}

Kamen Yotov, Keshav Pingali, Paul Stodghill,
{kyotov,pingali,stodghil}@cs.cornell.edu
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

ABSTRACT

Embedded processor designs are increasingly based on general-purpose processor families, modified and extended in various ways. However, the production of software for embedded processors remains a challenging problem. One promising approach for addressing this problem is *self-optimizing software*: instead of writing a program, one implements a program generator that produces a large number of program variants, and then determines empirically which variant performs best. The particular aspect of performance that is optimized can be execution time, power consumption, throughout, etc.

To prevent a combinatorial explosion in the number of program variants that have to be considered, self-optimizing systems bound the search space by exploiting knowledge of hardware parameters such as the number of registers, the capacity of the L1 cache, etc. For software to be truly self-optimizing, hardware parameter values relevant for software optimization must be determined automatically.

This paper makes the following contributions.

- We describe X-Ray – a robust and extensible micro-benchmark framework for measuring hardware parameters, in which it is very easy to implement new micro-benchmarks. This is particularly important in the embedded processor context because designers constantly add new features to architectures.
- We describe novel algorithms for measuring commonly used hardware parameters and show how they can be implemented in this framework. We evaluate our implementation experimentally on both embedded and desktop architectures, and show that it produces more accurate and complete results than existing tools.

^{*}This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140.

1. INTRODUCTION

The traditional approach of designing embedded processors by integrating independently-designed, non-programmable hardware blocks is giving way to designs based on general-purpose processor families such as the MIPS or Power architectures, extended or reduced in various ways. The new approach makes embedded processors more flexible because they operate under software control, which is critical for devices like Personal Digital Assistants (PDAs) that must run a variety of applications such as calendars, e-mail, etc.

However, the production of high-quality application software for programmable embedded processors remains a major challenge. Hardware vendors typically provide compilers, debuggers, etc., obtained either by modifying an existing tool chain for an established processor family or by writing them from scratch. Unfortunately, the quality of code produced by compilers even for conventional desktop processors can be far from optimal [15]; in the embedded processor context, this problem is exacerbated by other considerations such as power consumption. Programming in assembly language is often the only recourse.

One promising new technology for solving this problem is to write *self-optimizing* software that can improve its own performance without manual intervention. The key idea behind self-optimizing software is the *generate-and-test* paradigm: instead of writing a program, one writes a *program generator* that can generate an entire space of programs, and runs all these programs on the actual platform to determine which one performs best. The particular aspects of performance that are optimized may include time to completion of a task, the throughput of the entire system, power consumption, etc. Some self-optimizing systems such as FFTW [3, 7] and SPIRAL [4, 11], which generate optimized code for signal processing applications, optimize themselves statically when they are installed on the hardware platform, while other systems such as those envisioned in IBM's autonomic systems initiative [2] optimize their performance continuously during execution.

To control the combinatorial explosion in the number of program versions to be evaluated, self-optimizing systems use parameters of the hardware architecture to prune the search space. For example, the size of the L1 data cache influences the execution plans generated by FFTW in its FFT implementation. Other self-optimizing systems also exploit knowledge about the number of registers, instruction latencies, existence of certain instructions such as fused multiply-add (FMA), SIMD vector extensions, etc., to reduce the size of the search space [1, 14, 15].

For self-optimizing software, it is desirable that hardware parameter values relevant for software optimization be determined automatically without human intervention. It is important to note that these values are not necessarily the same as the values one might find in a hardware manual. For example, loop unrolling must take into account the number of registers available to hold values computed in the loop body. However, most compilers set aside certain registers for holding special values such as the stack or frame pointer, so the number of registers available to the register allocator is usually less than the total number of architected registers. In practice, it is hard enough to find documentation on hardware parameter values, let alone on the values relevant to software optimization.

In this paper, we describe X-Ray, a framework that uses micro-benchmarks to measure relevant values of hardware parameters automatically. Perhaps the most well-known micro-benchmark is the Saavedra benchmark [12, 8] that determines memory hierarchy parameters by measuring the time required to access a series of array elements with different strides. However, the timing results from this benchmark have to be analyzed manually to determine memory hierarchy parameters, so it is not suitable for incorporation into a self-optimizing system. Furthermore, this benchmark does not measure other processor parameters. As we discuss later, X-Ray does not suffer from these limitations.

For portability, X-Ray is entirely implemented in ANSI C [6]. One of the interesting challenges of this approach is to ensure that the C compiler does not perform any high-level restructuring optimizations on our benchmarks that might pollute the timing results, while performance critical optimizations, such as register allocation, are still enabled.

The rest of this paper is organized as follows. In Section 2, we give an overview of the framework and the techniques that it uses for generating and timing C programs. Sections 3 and 4 discuss the micro-benchmarks we have developed for measuring CPU and memory hierarchy parameters respectively. Since embedded systems typically do not have a deep memory hierarchy, we focus on L1 data cache parameters. We present experimental results in Section 5 and conclude in Section 6.

2. THE X-RAY FRAMEWORK

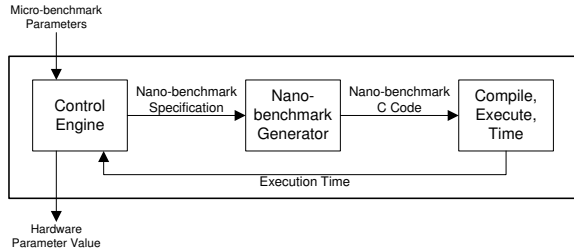


Figure 1: A micro-benchmark in X-Ray

Figure 1 shows the structure of a *micro-benchmark* in the X-Ray framework. Each hardware parameter is measured by one such micro-benchmark.

For example, consider measuring the number of available registers for a particular data type T . One way to determine this value is to execute a number of experiments, all of which perform the same computations, but on a different number of variables (N) of type T . When N exceeds the number of available registers for type T , not all variables can be register

allocated, and execution time should increase substantially. The number of available registers can be inferred from this cross-over point.

Some general conclusions can be drawn from this example. A single micro-benchmark may need to execute several timing experiments, which we call *nano-benchmarks*. Since there may be no *a priori* bound of the number of required nano-benchmarks, we need a *Nano-benchmark Generator*, which can produce *Nano-benchmark C Code* from a high-level *Nano-benchmark Specification*. Finally, generation should happen on-the-fly since the results of one nano-benchmark may determine the nano-benchmark to be executed next.

The execution of a micro-benchmark is orchestrated by its *Control Engine*, which chooses which nano-benchmarks to execute, along with the appropriate parameters, and the order in which to execute them. It determines the value of the hardware parameter based on these timing results.

Some micro-benchmarks may also require input from the results of running other micro-benchmarks. For example, to determine the latency of an instruction in cycles rather than in nanoseconds, the control engine needs to know the cycle time of the processor. This can be specified by the user or it can be measured by another micro-benchmark as discussed in Section 3.

2.1 Nano-benchmarks

Even with access to a high-resolution timer, it is hard to accurately time operations that take only a few CPU cycles to execute. Suppose we want to measure the time required to execute a C statement S . If this time is small compared to the granularity of the timer, we must measure the time required to execute this statement some number of times R_S (dependent on S), and divide that time by R_S . If R_S is too small, the time for execution cannot be measured accurately, whereas if R_S is too big, the experiment will take longer than it needs to.

```

 $R_S \leftarrow 1;$ 
while (measure $S$ ( $R_S$ ) <  $t_{min}$ )
   $R_S \leftarrow R_S \times 2;$ 
return (measure $S$ ( $R_S$ )  $\div$   $R_S$ );
  
```

Figure 3: High-level structure of a nano-benchmark

Figure 3 shows the timing strategy used in X-Ray nano-benchmarks. In this code, `measure S (R_S)` measures the time required to execute R_S repetitions of statement S . To determine a reasonable value for R_S , the code in Figure 3 starts by setting R_S to 1, and then doubles it until the experiment runs for at least t_{min} seconds. The value of t_{min} can be specified by the user and defaults to 0.25 seconds in the current implementation.

A simplistic implementation of `measure S` is shown in Figure 2(a). The call to `now()` is assumed to invoke the most accurate timer available on the platform. The code in Figure 2(a) incurs considerable loop overhead, which might even be greater than the time spent in executing S . To address this problem, we can unroll the loop U times as shown in Figure 2(b).

Another problem is that restructuring compiler optimizations may corrupt the experiment. For example, consider the case when we want to measure the latency of a single addition. In our framework, we would measure the time taken to execute the C statement $p_0 = p_0 + p_1$. It is impor-

```

measureS(R) {
  ts = now();
  i = R;
loop: S;
  if (--i)
    goto loop;
  te = now();
  return te - ts;
}
(a)

measureS(R) {
  ts = now();
  i = R / U;
loop:
  S;
  S;
  ...repeat U
times...
  S;
  if (--i)
    goto loop;
  te = now();
  return te - ts;
}
(b)

measureS(R) {
  initialize;
  volatile int v = 0;
  switch (v)
  {
    case 0:
      i = R/U;
      ts = now();
      loop:
      case 1: S1;
      case 2: S2;
      ...
      case i: Si;
      ...
      case n: Sn;
      case n+1: S1;
      ...
      case W: Sn;
      if (--i)
        goto loop;
      te = now();
      if (!v)
        return te - ts;
  }
  use;
}
(c)

measureS(R) {
  initialize;
  volatile int v = 0;
  switch (v)
  {
    case 0:
      i = R/U;
      ts = now();
      loop:
      case 1: S1;
      case 2: S2;
      ...
      case i: Si;
      ...
      case n: Sn;
      case n+1: S1;
      ...
      case W: Sn;
      if (--i)
        goto loop;
      te = now();
      if (!v)
        return te - ts;
  }
  use;
}
(d)

```

Figure 2: Implementation of `measureS`

tant to optimize p_0 and p_1 by allocating them in registers, but it is crucial not to optimize the U statements in the loop body to $p_0 = p_0 + U \times p_1$, which would prevent from timing the original statement correctly.

To solve such problems, we need to generate programs which the compiler can aggressively optimize, without disrupting the sequence and type of operations, whose execution time we want to measure. We solve this problem using a `switch` statement on a `volatile` variable v as shown in Figure 2(c). The semantics of C require that all accesses of v go to the memory hierarchy, and therefore the compiler cannot assume anything about which `case` of the `switch` is selected. Because there is potential control flow to each of the `case` blocks, it is impossible to combine or reorder them in any way.

The final problem is that if the compiler is able to deduce that the result of the computations performed in S is not used in the rest of the code, it might perform dead-code elimination and remove all instances of S altogether. To prevent this unwanted optimization we assign the initial value of all variables that appear in S from appropriately typed `volatile` variables in the `initialize` statement and copy their final values back to the same `volatile` variables in the `use` statement.

As we will see in Section 3, there are cases where we wish to measure the performance of a sequence of different statements S_1, S_2, \dots, S_n . To prevent the compiler from optimizing this sequence, the code generator will give each S_i a different case label, generating code of the form shown in Figure 2(d). In this figure, the number of case labels W is the smallest multiple of n greater than or equal to U .

2.2 Nano-benchmark Generator

The X-Ray nano-benchmark generator accepts as an input

a nano-benchmark specification and produces nano-benchmark C code structured as shown in Figure 2(d).

The nano-benchmark specification is a tuple which contains a statement S to be timed and type information for all variables in S . For example, to measure the latency of double-precision floating point ADD operation, we use the nano-benchmark specification $\langle p_1 = p_1 + p_2, \langle p_1, p_2 : \text{F64} \rangle \rangle$, which means that we time the statement $p_1 = p_1 + p_2$, where p_1 and p_2 are variables of type double (defined as F64 in X-Ray). Given this specification, the nano-benchmark generator can produce code as shown in Figure 2(c). Generating code of the form shown in Figure 2(d) is more complex and requires the first element of the tuple to be a function $f : \text{integer} \rightarrow \text{string}$, which computes the code for statement S_i from the case label i .

2.3 Implementing a new micro-benchmark

The following two steps describe the process of implementing a new micro-benchmark within the X-Ray framework.

1. Decide what timing experiments are needed and implement the corresponding nano-benchmarks. If the structure of these nano-benchmarks fits the output of the X-Ray nano-benchmark generator, as presented in Figure 2(d), one only needs to provide the corresponding nano-benchmark specifications.
2. Implement the micro-benchmark control engine, describing which nano-benchmarks to run, with what parameters, in what order, and how to produce a final result from the external parameters and the timings.

As we will see in Section 3, both these steps require very few lines code.

3. CPU MICRO-BENCHMARKS

We now describe how X-Ray measures a number of key CPU parameters. X-Ray can also measure the degree of SMT and SMP thread-level parallelism, but we do not discuss it here because it is not relevant in the context of embedded processors.

3.1 CPU Frequency

CPU frequency (F_{CPU}) is an important hardware parameter because other parameters are measured relative to it (in clock cycles). X-Ray assumes that dependent integer additions can be executed at the rate of one per cycle, which is valid for most current processors. The assumption of dependence is important because modern architectures can often issue two or more independent integer addition operations in one cycle, so timing independent addition operations would be misleading.

For this micro-benchmark we use a single nano-benchmark with specification $S = \langle p_0 = p_0 + p_1, \langle p_0, p_1 : \text{int} \rangle \rangle$. Given the time $\text{time}(S)$ in nanoseconds required to execute the statement S , we compute the CPU frequency in MHz as follows.

$$F_{\text{CPU}} \leftarrow 1000 \div \text{time}(S)$$

3.2 Instruction Latency

The latency $L_{O,T}$ of an operation (instruction) O , with operands of type T , is the number of cycles after one such instruction is dispatched until its result becomes available to subsequent dependent instructions.

For this micro-benchmark we use a nano-benchmark with specification $S_{O,T} = \langle p_0 = O(p_0, p_1), \langle p_0, p_1 : T \rangle \rangle$. We then compute the instruction latency in clock cycles as follows.

$$L_{O,T} \leftarrow \frac{\text{time}(S_{O,T})}{1000 \div F_{\text{CPU}}}$$

3.3 Instruction Throughput

The throughput $TP_{O,T}$ of an operation (instruction) O , with operands of types T , is the rate in cycles at which the CPU can issue independent instructions of that type. On modern processors the throughput of an instruction is usually much smaller than its latency, because of pipelining and super-scalar execution.

To measure $TP_{O,T}$, we use a nano-benchmark specification $S_{O,T,N} = \langle p_i \% N = O(p_i \% N, p_N), \langle p_0, p_1, \dots, p_N : T \rangle \rangle$. Note that this specification generates code of the form shown in Figure 2(d). It is further parameterized by N – the number of independent instructions to generate. For example, the sequence of statements generated for $N = 3$ is the following.

```

case 0 :  $p_0 = O(p_0, p_3)$ ;
case 1 :  $p_1 = O(p_1, p_3)$ ;
case 2 :  $p_2 = O(p_2, p_3)$ ;
case 3 :  $p_0 = O(p_0, p_3)$ ;
...
case  $\bar{W}$  :  $p_2 = O(p_0, p_3)$ ;

```

We then compute the instruction throughput in clock cycles as follows.

$$\begin{aligned}
& N \leftarrow 2; \\
& \text{while} \left(\frac{\text{time}(S_{O,T,N})}{\text{time}(S_{O,T,N-1})} > 1 - \epsilon \right) \\
& \quad N \leftarrow N + 1; \\
& TP_{O,T} \leftarrow \frac{\text{time}(S_{O,T,N-1})}{1000 \div F_{\text{CPU}}};
\end{aligned}$$

The nano-benchmark code for $S_{O,T,N}$ exhibits instruction-level parallelism (ILP) on the order of N . The control engine times the nano-benchmark for successively growing values of N while performance continues to increase due to the additional ILP. When the performance levels off for some N , the throughput $TP_{O,T}$ is computed as the last (fastest) timing divided by the clock cycle time.

In practice, the X-Ray implementation of the throughput micro-benchmark is somewhat more complex. The reason for this increased complexity is that the implementation we just described does not permit the exploitation of instruction-level parallelism in statically scheduled VLIW cores because the case labels prevent the compiler from scheduling VLIW bundles with more than one instruction. Therefore we use a nano-benchmark specification with a more complex statement generating function, that puts B independent instructions at each case statement, as follows.

$$\begin{aligned}
& S_{O,T,N,B} = \\
& < \{ \\
& \quad p_{(i \times B + 0) \% N} = O(p_{(i \times B + 0) \% N}, p_N); \\
& \quad p_{(i \times B + 1) \% N} = O(p_{(i \times B + 1) \% N}, p_N); \\
& \quad \dots \\
& \quad p_{(i \times B + B - 1) \% N} = O(p_{(i \times B + B - 1) \% N}, p_N); \\
& \}, \\
& \langle p_0, p_1, \dots, p_N : T \rangle \\
& >
\end{aligned}$$

The corresponding control engine algorithm is as follows.

$$\begin{aligned}
& N \leftarrow 2; \\
& \text{while} \left(\frac{\text{time}(S_{O,T,N,1})}{\text{time}(S_{O,T,N-1,1})} > 1 - \epsilon \right) \\
& \quad N \leftarrow N + 1; \\
& B \leftarrow 2; \\
& N \leftarrow N - 1; \\
& \text{while} \left(\frac{\text{time}(S_{O,T,N \times B, B}) \div B}{\text{time}(S_{O,T,N \times (B-1), B-1}) \div (B-1)} > 1 - \epsilon \right) \\
& \quad B \leftarrow B + 1; \\
& TP_{O,T} \leftarrow \frac{\text{time}(S_{O,T,N \times (B-1), B-1}) \div (B-1)}{1000 \div F_{\text{CPU}}};
\end{aligned}$$

3.4 Instruction Existence

Many embedded processors do not have dedicated floating-point hardware. Some have single-precision floating-point functional units, but not double-precision ones. In case dedicated floating-point hardware is not present, usually an emulation library is used. In X-Ray we can easily determine the presence of dedicated floating-point hardware by measuring the latency of a floating-point ADD operation of the appropriate type. If the latency is more than a few cycles (10 in our implementation), we conclude that the operation is emulated, otherwise we conclude that it is executed in hardware.

In certain cases, it is not obvious how a certain C statement is translated to instructions. One very common operation for numerical applications is $p_1 = p_1 + p_2 \times p_3$. On some platforms, this statement is compiled into a single fused multiply-add instruction (FMA), while on some it is compiled into a separate multiply and add instructions. If an FMA instruction does not exist, the compiler will need an extra register to store the intermediate value and schedule two instructions instead of one. This has an impact on how such sequences of statements need to be scheduled. For

example, the ATLAS [1, 14] library generator produces different code for the compiler depending upon the existence of an FMA instruction.

With an FMA instruction the CPU can execute an add instruction “for free” together with a multiply instruction. Therefore we determine the existence of FMA by comparing the throughput of a simple multiply with that of a fused multiply-add.

3.5 Number of Registers

To measure the number of registers NR_T of particular type T , we use a single nano-benchmark with specification $S_{T,N} = \langle p_i \% N = p_i \% N + p_{(i+N-1)\%N}, \langle p_0, p_1, \dots, p_N : T \rangle \rangle$. For example, the sequence of statements generated for $N = 4$ is as follows.

```

case 0 :  $p_0 = p_0 + p_3$ ;
case 1 :  $p_1 = p_1 + p_0$ ;
case 2 :  $p_2 = p_2 + p_1$ ;
case 3 :  $p_3 = p_3 + p_0$ ;
case 4 :  $p_0 = p_0 + p_3$ ;
...
case W :  $p_3 = p_3 + p_0$ ;

```

If all of p_i are allocated in registers, the time per operation is much smaller than when some are allocated in memory. The goal is to determine the maximum N , for which no variables are allocated to memory. The control engine doubles N until it observes a drop in performance. After that it performs a binary search in the interval $[N \div 2, N)$. The actual control engine algorithm is as follows.

```

 $N \leftarrow 4$ ;
while ( $\frac{\text{time}(S_{T,N})}{\text{time}(S_{T,2})} < 1 + \epsilon$ )
     $N \leftarrow N \times 2$ ;
 $R \leftarrow N$ ;
 $L \leftarrow \frac{N}{2}$ ;
while ( $R - L > 1$ )
     $P \leftarrow \frac{(R+L)}{2}$ ;
    if ( $\frac{\text{time}(S_{T,P})}{\text{time}(S_{T,2})} < 1 + \epsilon$ )
         $R \leftarrow P$ ;
    else
         $L \leftarrow P$ ;
 $NR_T \leftarrow L$ ;

```

4. CACHE MICRO-BENCHMARKS

In this section, we present algorithms for automatically measuring memory hierarchy parameters that are important for software. Although X-Ray can measure parameters of all levels of cache and the TLB, we will focus on the L1 cache because most embedded processors do not have an L2 cache. Complete description of X-Ray’s memory hierarchy measurement capabilities is given in [16].

The most well-known benchmark for measuring memory hierarchy parameters is the Saavedra benchmark [12], which measures the time required to access array elements with different strides. The timing results of this benchmark are usually inspected manually to determine memory hierarchy parameters. In contrast, the X-Ray micro-benchmarks described in this section produce the values of these parameters directly. Moreover, our algorithm for measuring cache capacity is the first one that is designed to accurately work

for caches whose capacity is not a power of 2 (found on the Itanium 2 for example), and for caches that support exclusion, such as the caches on AMD machines. Finally, our benchmarks, unlike existing benchmarks, are not affected by hardware pre-fetching.

4.1 Cache Parameters

We will focus on the *associativity*, *block size*, *capacity*, and *hit latency* [8] of caches. The first three parameters are sometimes referred to as the $\langle A, B, C \rangle$ of caches. Our experiments assume that the replacement policy is least-recently-used (LRU), since almost all caches implement variants of this policy, and our experiments show that even when they do not, the results are still accurate.

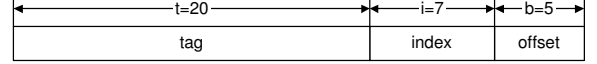


Figure 4: Memory address decomposition on P6

Figure 4 shows the typical structure of a memory address. We use the Intel P6 (Pentium Pro/II/III) architecture in the following explanations. On these machines, the L1 data cache is organized as $\langle A, B, C \rangle = \langle 4, 32, 16\text{KB} \rangle$. Therefore the cache contains $C \div B = 16384 \div 32 = 512$ individual blocks, divided into $512 \div A = 512 \div 4 = 128$ sets of 4 blocks each. The highest $t = 20$ bits constitute the block **tag**, $i = 7$ bits are needed to **index** one of the 128 sets, and $b = 5$ bits are needed to store the **offset** of a particular byte within the 32-byte block.

DEFINITION 1. For a cache with associativity A and capacity C , we define the stride T of that cache as $T \equiv \frac{C}{A}$.

Note that $T = 2^{i+b}$, and thus $C = A \times 2^{i+b}$. Unlike cache stride, associativity and capacity do not have to be a power of 2. For example, some versions of Intel Itanium 2 have a 24-way set associative L3 cache with capacity 6MB.

4.2 Sequences and compact sequences

When all addresses of an address sequence W can coexist together in a cache we say that W is *compact* with respect to that cache and the average access time is the cache hit latency l_{hit} . When the sequence is not compact and we repeatedly access its elements the cache will suffer some misses. If every single access is a cache miss, we say that W is *non-compact* and the average access time is the cache miss latency l_{miss} , which is typically much greater than l_{hit} . Finally, when some accesses are cache hits and some are cache misses, the average access time is between l_{hit} and l_{miss} and we say that W is *semi-compact*.

Some of our micro-benchmarks access sequences of N addresses, where successive addresses are separated by a stride $S = 2^\sigma$. Such sequences are completely characterized by their starting address m_0 , stride S and number of elements N and therefore we use the notation $\langle m_0, S, N \rangle$ to represent them.

Theorem 1 describes the necessary and sufficient conditions for compactness and non-compactness of a sequence of this type for a given cache. Informally, this theorem says that as the stride S gets bigger, the maximum length of a compact sequence with that stride decreases until it bottoms out at A , while the minimum length of a non-compact sequence with that stride decreases until it bottoms out at $A + 1$.

THEOREM 1. Consider a cache with parameters $\langle A, B, C \rangle$ and a sequence $W = \langle m_0, S, N \rangle$.

- (a) *compact*(W) $\Leftrightarrow N \leq N_c = A \lceil \frac{T}{S} \rceil$
- (b) *non-compact*(W) $\Leftrightarrow N \geq N_{nc} = (A + 1) \lceil \frac{T}{S} \rceil$

PROOF. Omitted. See [16] \square

4.3 Algorithms for Measuring L1 Cache Parameters

In this section we use *is_compact*(W) to empirically determine if W is compact. Our implementation of this function repeatedly accesses each address in W , computes the average time per access l , and declares the sequence to be compact if l is close to the hit latency of the cache l_{hit} , which is measured as described in Section 4.3.1.

Although this procedure seems simple in principle, the timing measurements require some special care as discussed in Section 4.4.

4.3.1 Cache Latency

We determine l_{hit} by measuring the average time per access of the sequence $\langle m_0, 1, 1 \rangle$, which is compact since it contains a single element.

4.3.2 Capacity and Associativity

Theorem 1 suggests a method for determining the capacity C and the associativity A of the cache. First, we find A by determining the asymptotic limit of the length of a compact sequence as the stride is increased. The smallest value of the stride for which this limit is reached is T , the stride of the cache; once we know A and T , we can find C .

```

 $S \leftarrow 1;$ 
 $N \leftarrow 1;$ 
while (is_compact( $\langle m_0, S, N \rangle$ ))
   $N \leftarrow 2 \times N$ 
 $N_{old} \leftarrow N;$ 
 $N \leftarrow 0;$ 
while ( $N \neq N_{old}$ )
   $S \leftarrow 2 \times S;$ 
   $N_{old} \leftarrow N;$ 
   $N \leftarrow \min N_{min} \in [1, N_{old}] : \neg \text{is\_compact}(\langle m_0, S, N_{min} \rangle);$ 
 $A \leftarrow N - 1;$ 
 $C \leftarrow \frac{S}{2} \times A;$ 

```

Figure 5: Measuring C and A of L1 Data Cache

Pseudo-code for measuring C and A of the L1 data cache is shown in Figure 5. The algorithm can be described as follows. Start with the sequence $\langle m_0, S, N \rangle = \langle m_0, 1, 1 \rangle$, which is compact, and keep doubling N until the sequence is not compact. Let N_{old} be the first N for which this happens. Now start doubling the stride S , and for each S compute the smallest N , for which $\langle m_0, S, N \rangle$ is not compact. This value of N can be found by using binary search in the interval $[1, N_{old}]$. If $N \neq N_{old}$, Let $N_{old} = N$ and recompute N for the next S . Repeat this step until $N = N_{old}$. At this point, declare $A = N - 1$ and the $C = \frac{S}{2} \times A$.

Note that the number of addresses accessed by the algorithm in this micro-benchmark is on the order of the associativity of the cache, which is superior to previous approaches because non-compactness produces a very pronounced performance drop, which is much easier to detect automatically.

4.3.3 Block Size

For given cache parameters C , A and T , $\langle m_0, T, 2A \rangle$ is non-compact since all $2A$ addresses map to the same cache set. This sequence can also be expressed as $\langle m_0, T, A \rangle \cup \langle m_0 + C, T, A \rangle$. If we offset the second half of the sequence by a constant δ , as shown in Figure 6, we get $\langle m_0, T, A \rangle \cup \langle m_0 + C + \delta, T, A \rangle$.

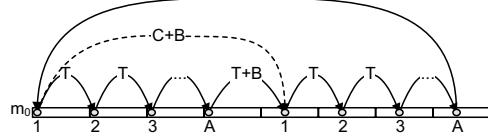


Figure 6: Address sequence for measuring B

The addresses of the two subsequences map to a single cache set. When $0 \leq \delta < B$ this cache set is the same. When $\delta \geq B$ this cache set is unique to each of them. Therefore the smallest value of δ for which the full sequence is compact is $\delta = B$. Figure 7 shows pseudo-code for the algorithm.

```

 $\delta \leftarrow 1$ 
while ( $\neg \text{is\_compact}(\langle m_0, T, A \rangle \cup \langle m_0 + C + \delta, T, A \rangle)$ )
   $\delta \leftarrow 2 \times \delta;$ 
return  $\delta;$ 

```

Figure 7: Algorithm for measuring B

4.4 Implementation of *is_compact*

In this section we describe some of the important implementation details of the implementation of *is_compact*(W).

The array of elements is declared of type pointer (`void *`) instead of integer (`int`) as in the Saavedra benchmark. The array is initialized in such a way that each element contains the address of the element which should be accessed immediately after it. A local variable p is initialized with the address of the element which should be accessed first.

For a correct implementation of *is_compact*(W), it is important that we repeatedly access all elements of the sequence, but the actual order in which we access them is irrelevant. To prevent hardware constant stride prefetchers, like those on the IBM Power architecture, from interfering with our timings, we initialize the array elements by chaining the pointers so that we visit the elements in a pseudo-random order.

Suppose the address sequence is m_0, m_1, \dots, m_{n-1} . One way to reorder this sequence is to choose a number p , such that p and n are mutually prime. Then, after element m_i , visit element $m_{(i+p) \bmod n}$ instead of element $m_{(i+1) \bmod n}$. As p and n are mutually prime, the recurrence $i \leftarrow (i + p) \bmod n$ is guaranteed to generate all the integers between 0 and $n - 1$ before repeating itself.

To perform the timings we use a nano-benchmark with specification $\langle p = *(\text{void} **), p, \langle p : \text{void} * \rangle \rangle$. The fact that we can use the same nano-benchmark generator for measuring both CPU parameter values and cache parameter values demonstrates the flexibility of the X-Ray architecture.

5. EXPERIMENTAL RESULTS

In this section we present experimental results obtained by using X-Ray to measure the hardware parameters of the following embedded and desktop processors.

- ARM SA1110, 200MHz

- xScale PXA250, 400MHz
- Xtensa LX, 350MHz
- MIPS R4400, 150MHz
- Power 3, 375MHz
- Pentium 4 Xeon, 3.06GHz
- Itanium 2, 1.5GHz

Although X-Ray is able to automatically determine a large number of hardware parameters, we only discuss parameters that are most relevant for embedded processors. Therefore we do not discuss parameters of L2 and lower cache levels, double-precision floating point operations, symmetric multiprocessing, and simultaneous multi-threading, etc. that X-Ray is capable of measuring.

We compare our results to the actual values of the hardware parameters, as well as to the values obtained by `lmbench v3.0-a4` [9, 10, 13]. `lmbench` is a well-known suite of benchmarks for measuring operating systems parameters such as thread-create time and context-switch time. In the latest version, the authors have included several hardware benchmarks for measuring CPU frequency, latency and parallelism of different operations, capacity, block size, and latency of each level of the memory hierarchy, and the number of TLB entries.

A summary of the experimental results is presented in Table 1. Because both `lmbench` and X-Ray measure hardware parameters empirically, the results vary somewhat from one execution to the next. However, these variations are fairly small, so the results we present are from a single run.

X-Ray and `lmbench` measure some hardware parameters in different units. To allow direct comparison, we normalized `lmbench` results as follows.

- `lmbench` measures the processor clock cycle c_{lmb} and various latencies l_{lmb} in nanoseconds. We compute the processor frequency in MHz as $f_{lmb} = 1000 \div c_{lmb}$, and latency in cycles as $l_{lmb} \div c_{lmb}$.
- Instead of measuring instruction throughput, `lmbench` measures available instruction parallelism p_{lmb} . We compute instruction throughput in cycles as $t_{lmb} = l_{lmb} \div p_{lmb}$, where l_{lmb} is the latency in cycles of the corresponding instruction, computed as shown above.

We now discuss some of the more interesting results for each platform.

5.1 ARM SA1110 and xScale PXA250

We ran our ARM SA1110 experiments on an HP iPAQ PDA, running Intimate Linux 2.4.18-rmk3, using GCC v2.95.4. For xScale PXA250 we used an HP iPAQ H3900 PDA, running Familiar Linux 2.4.19-rmk6-pxa1-hh37, using GCC v3.4.3.

Unfortunately, we were unable to build `lmbench` on these platform and we are communicating with the authors to resolve the problem. If possible, we will present the `lmbench` results in the camera-ready version of the paper.

As Table 1 shows, X-Ray measured accurately all parameters of interest. Moreover, X-Ray successfully determined that there is no floating point hardware on any of these two hardware platforms. The number of measured I32 registers is smaller than the actual number, because the compiler and/or the hardware allocate some of the registers with special purpose (stack pointer, frame pointer, etc.)

5.2 Xtensa LX

Xtensa LX is a configurable, extensible processor core, designed by Tensilica, Inc [5]. They provide a generator that, given a processor configuration, can automatically build a processor core design along with a cycle-accurate processor simulator and a development tool chain. Therefore, the hardware parameters of any Xtensa LX core instance could be different by construction. This feature of the Xtensa LX processor makes it an excellent target for a micro-benchmark framework like X-Ray, which allows intelligently designed software to automatically adapt and optimize itself for the diverse features of the processor.

We experimented with different processor configurations and ran X-Ray using the corresponding cycle-accurate simulators. We still have problems with running `lmbench` and its results will be presented in the final paper, if these issues are resolved.

Because we ran X-Ray directly without an underlying operating system, we could carefully analyze even the minor inaccuracies.

5.2.1 Frequency

The processor frequency measured by X-Ray (343.225MHz) was 2% off of the actual value (350MHz). This inaccuracy can be explained with the loop overhead incurred from the code shown in Figure 2(d). The 256 unrolled case statements have a latency of 1 cycle for a total of 256 cycles, and the loop-back code at the end has a latency of 5 cycles (1 cycle for the decrement of `i`, one cycle for the branch instruction, and 3 cycles penalty for non-aligned branch target). Therefore the measurement error is $5 \div 261 \approx 2\%$. While we can partially compensate this error [13], we do not feel that this is necessary, because we only use frequency to measure other parameters relative to it (in clock cycles), and the incurred error is similar in all measurements.

5.2.2 Number of Registers

X-Ray measured only 11 integer registers, while 16 are architecturally available. We verified that register spills were happening when using 12 registers in our measurement sequence, because the rest were used by the compiler. On the other hand, X-Ray measured 17 single precision floating point registers, while only 16 are architecturally available. We examined the generated assembly source and indeed, when using a measurement sequence of length 17, there was one register spill. The reason X-Ray did not detect this spill was that our edge-detection threshold was too big and the data cache access incurred by the spill was not costly enough to be measured. Of course, we could fix our threshold, but this might have a negative impact on other platforms, where measurement noise is relatively high. In practice, the fact that the data cache accesses went unnoticed in X-Ray means that when optimizing real applications, the performance penalty will not be statistically significant even if a few more variables are used than the number of available registers.

5.2.3 Other Configurations

We took advantage of the possibility to configure the Xtensa LX core differently and see if X-Ray is able to accurately measure the new parameters. We tried the following three changes to the original configuration.

- We introduced two additional integer ADD and one

Feature	Tool	ARM SA1110	xScale PXA250	Xtensa LX	R4400	Power 3	Pentium 4	Itanium 2
Frequency (MHz)	Actual	200.000	400.000	350.000	150.000	375.000	3,060.000	1,500.000
	X-Ray	203.586	393.489	343.225	145.889	375.434	6,097.130	1,488.344
	Imbench	!compiled	!compiled	!compiled	152.001	374.953	3,049.710	1,497.903
Latency ADD I32 (cycles)	Actual	1.000	1.000	1.000	1.000	1.000	0.500	1.000
	X-Ray	1.001	1.006	1.000	1.000	1.000	1.001	1.000
	Imbench	!compiled	!compiled	!compiled	1.075	1.009	0.488	1.004
Latency MULTIPLY I32 (cycles)	Actual	3.000	2.000	2.000	12.000	3.000	14.000-18.000	4.000
	X-Ray	3.095	1.977	1.977	14.873	3.000	29.987	3.985
	Imbench	!compiled	!compiled	!compiled	15.726	3.007	13.907	5.168
Throughput ADD I32 (cycles)	Actual	1.000	1.000	1.000	1.000	0.500	0.250	0.167
	X-Ray	1.001	1.003	0.996	1.000	0.497	0.679	0.169
	Imbench	!compiled	!compiled	!compiled	0.814	0.490	0.375	0.469
Throughput MULTIPLY I32 (cycles)	Actual	2.000	1.000	1.000	12.000	3.000	5.000	5.000
	X-Ray	1.986	0.998	0.996	14.870	2.972	9.337	0.506
	Imbench	!compiled	!compiled	!compiled	13.441	3.007	3.512	0.485
NR I32 (count)	Actual	16	16	16	32	32	8	128
	X-Ray	12	13	11	17	28	5	123
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
FMA I32 (boolean)	Actual	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE
	X-Ray	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
FPU F32 (boolean)	Actual	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
	X-Ray	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
Latency ADD F32 (cycles)	Actual	n/a	n/a	4.000	4.000	4.000	5.000	4.000
	X-Ray	n/a	n/a	3.927	3.935	4.000	10.013	3.984
	Imbench	!compiled	!compiled	!compiled	4.179	4.012	4.849	4.284
Latency MULTIPLY F32 (cycles)	Actual	n/a	n/a	4.000	7.000	4.000	7.000	4.000
	X-Ray	n/a	n/a	3.927	6.870	5.034	14.079	3.984
	Imbench	!compiled	!compiled	!compiled	7.323	4.014	6.892	4.014
Throughput ADD F32 (cycles)	Actual	n/a	n/a	1.000	3.000	0.500	1.000	0.500
	X-Ray	n/a	n/a	0.996	2.913	0.497	2.030	0.506
	Imbench	!compiled	!compiled	!compiled	2.943	0.501	2.108	0.549
Throughput MULTIPLY F32 (cycles)	Actual	n/a	n/a	1.000	3.000	0.500	2.000	0.500
	X-Ray	n/a	n/a	0.996	2.957	0.500	4.004	0.506
	Imbench	!compiled	!compiled	!compiled	3.130	0.501	2.209	0.515
NR F32 (count)	Actual	n/a	n/a	16	32	32	8	128
	X-Ray	n/a	n/a	17	24	32	8	128
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
FMA F32 (boolean)	Actual	n/a	n/a	TRUE	FALSE	TRUE	FALSE	TRUE
	X-Ray	n/a	n/a	TRUE	FALSE	TRUE	FALSE	TRUE
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
FPU F64 (boolean)	Actual	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
	X-Ray	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
L1 Cache Capacity (KB)	Actual	8	32	16	16	64	8	16
	X-Ray	8	32	16	16	64.5	8	16
	Imbench	!compiled	!compiled	!compiled	16	64	8	16
L1 Cache Associativity (count)	Actual	32	32	2	1	128	4	4
	X-Ray	32	32	2	1	129	4	4
	Imbench	!compiled	!compiled	!compiled	!supported	!supported	!supported	!supported
L1 Cache Block Size (bytes)	Actual	32	32	64	16	128	64	64
	X-Ray	32	32	64	16	128	64	64
	Imbench	!compiled	!compiled	!compiled	16	128	64	64
L1 Cache Latency (cycles)	Actual	2.000	3.000	2.000	3.000	2.000	2.000	2.000
	X-Ray	1.993	3.053	1.969	2.956	1.986	4.109	2.009
	Imbench	!compiled	!compiled	!compiled	3.140	2.023	2.226	2.010
Main Memory Latency (cycles)	Actual	?	?	?	?	?	?	?
	X-Ray	41.259	134.986	10.793	13.820	18.126	36.747	6.024
	Imbench	!compiled	!compiled	!compiled	15.060	18.317	17.963	6.030

Table 1: Summary of Experimental Results

additional integer MULTIPLY functional units. X-Ray correctly measured the new Throughput ADD I32 and Throughput MULTIPLY I32 as 0.335 and 0.496 cycles respectively.

- We changed the data cache configuration to 6KB, 3-way set associative with 32 byte blocks. X-Ray correctly measured the new cache parameters.
- We replaced the data cache with a 128-bit single pre-

cision fixed point SIMD unit (Vectra LX). After the appropriate descriptions were added, X-Ray correctly measured the latency and throughput of ADD and MULTIPLY, along with the number of vector registers (1.000, 1.977, 0.998, 0.996, and 16 respectively).

5.3 MIPS R4400

Although MIPS R4400 is used as an embedded processor today, we were unable to find embedded hardware on which

to perform our experiments. Instead, we used an old SGI Workstation, running IRIX 6.2 and the native MIPSPro C Compiler.

On this platform, X-Ray and lmbench gave similar and relatively accurate results. X-Ray measured all hardware parameters except the frequency more accurately than lmbench. Furthermore, in addition to the parameters measured by lmbench, X-Ray also measured the number of registers of various types, the existence of single and double precision floating point hardware and a specialized fused multiply-add instruction, and the cache associativity.

There are two details worth noting.

- The latency of MULTIPLY I32 measured by X-Ray is about 15 cycles, while the actual latency is 12 cycles. The reason behind this mismatch is that the R4400 has special registers `hi` and `lo`, which hold the result of integer multiply. Therefore the code sequence we use (`r0=r0*r1`) is translated to the assembly sequence `<hi,lo> = r0 * r1; r0 = lo; noop; noop`. The two `noop` instructions are necessary because access to `lo` is asynchronous and the compiler needs to make sure that the value can be copied before it is destroyed. This measurement inconsistency can be looked at as a feature, when generating code for R4400, because although the latency of an integer multiply is 12 cycles, it cannot be sustained by code. Moreover X-Ray determined that there exists an integer fused multiply-add instruction, while in fact there is not one. Again, this was possible, because an ADD instruction can be included for free by the compiler in place of one of the `noop` instructions, which follow a MULTIPLY.
- X-Ray measured significantly fewer registers than are architecturally available. We examined the generated assembly files and confirmed that it is the policy of the native compiler to reserve the rest of the registers. In particular, when measuring floating-point registers, a sequence of length 24 incurred no spills. Using a sequence of 25 incurred a spill for one variable, and one of the 8 reserved registers was used by load operations to load this variable from memory. Therefore 24 is the correct number of registers for allocation of variables.

5.4 Power 3

Although embedded platforms with Power 3 processors are common, we were not able to obtain embedded platform using the PowerPC ISA. Therefore we used an IBM Power 3 Workstation, running AIX and using the VisualAge for C v5 compiler.

There are several details worth mentioning:

- X-Ray detected an integer fused multiply-add instruction although there is not one in the ISA. We verified that even though our measurement sequence (`r0=r0+r0*r0`) is translated into separate dependent MULTIPLY and ADD instructions, the hardware can achieve the same throughput as if there was no ADD. Therefore such sequence is useful in generating high-performance code for this architecture.
- X-Ray measured the data cache as 129-way set associative instead of 128-way set associative. This resulted in capacity of 64.5KB compared to the documented

one of 64KB. We investigated the result by hand and verified that the edge appears after 129, as opposed to 128. We still have not explained this phenomenon, but the numbers prove that no performance will be lost if code is optimized as if the cache is 129-way set associative.

Lmbench gave satisfactory results on the parameters it is able to measure.

5.5 Pentium 4 Xeon

Apart from experimenting with embedded processors, we ran X-Ray on a number of workstation and server-class machines. Here we present some of the interesting results for Pentium 4 Xeon. We used a machine running RedHat Enterprise Linux 2.4.21-27.0.1.ELsmp, along with GCC 3.2.3.

These processors feature two double-pumped integer ALUs, which led X-Ray to believe that the frequency is twice higher than the actual. Again, this is not a problem, as long as all other timings are measured relative to this frequency. Indeed, as Table 1 shows, all timing values measured by X-Ray are twice larger than the actual values (except for Throughput ADD I32).

The problem with Throughput ADD I32 is quite interesting. Because of the two integer ALUs and integer ADD latency of 0.5 cycles, we expect an effective throughput of 0.25 cycles, which translates to 0.5 cycles relative to our frequency. Instead X-Ray measured 0.679, which is 50% more (3 integer adds per cycle instead of 4). This problem occurs because the instruction cache on Pentium 4 can only deliver 3 instructions per cycle to the processor dispatch engine, therefore prohibiting the integer pipes to sustain maximum throughput. Once again, such *effective* as opposed *actual* parameter values are what are important in scheduling high-performance code.

Lmbench results are close to those of X-Ray but noticeably less accurate. The greatest error is for Throughput MULTIPLY I32 and is close to 30%. Lmbench managed to find the advertised frequency instead of the double value as X-Ray.

Although we do not present the results here, it is worth noting that X-Ray was able to accurately measure the number of vector registers (MMX, SSE, and SSE2), as well as the latencies and throughputs of the corresponding SIMD instructions.

5.6 Itanium 2

For our Itanium 2 experiments we used a server machine running SUSE Linux 2.4.18-e.31smp, along with GCC 2.96. X-Ray produced accurate results for all parameters. Lmbench results were slightly less accurate, with one major problem – Throughput ADD I32. This processor is able to execute 6 independent ADD operations per cycle, and lmbench measured throughput of only 0.469. X-Ray measured the correct throughput of 0.169.

6. CONCLUSIONS AND FUTURE WORK

This paper presented the X-Ray framework, which is able to automatically measure a variety of important hardware parameters such as the existence, latency and throughput of different instructions, the number of registers, the parameters of the memory hierarchy, etc. The values measured are the values relevant to software running on the platform,

which may be different in general from the values of these parameters in hardware manuals.

We are actively developing new micro-benchmarks inside the X-Ray framework. Our current focus, and ideas for future work include:

- measuring the parameters of instruction caches,
- measuring other parameters of the memory hierarchy, like bandwidth and parallelism at each level, cache replacement policy and write mode,
- studying the effects of victim caches on the presented algorithms and resolving any issues that may arise, and
- designing a strategy for finding all bundles of instructions that can be issued in a single CPU cycle at a sustained rate.

X-Ray is freely available for non-commercial use at <http://iss.cs.cornell.edu/Software/X-Ray.aspx>.

7. ACKNOWLEDGEMENTS

We would like to thank Tensilica, Inc., and more specifically Darin Petkov, for performing the experimental evaluation of X-Ray on their Xtensa LX configurable, extensible processor core. He also suggested a number of improvements to the implementation of the framework.

We also want to thank to the people supporting www.handhelds.org for offering free access to their HP iPAQ PDA cluster. Without it, our experimental evaluation on ARM SA1110 would not have been possible.

8. REFERENCES

- [1] Automatically Tuned Linear Algebra Software (ATLAS). <http://math-atlas.sourceforge.net/>.
- [2] Autonomic Computing: creating self-managing computing systems. <http://www.research.ibm.com/autonomic/>.
- [3] FFTW: Fastest Fourier Transform in the West. <http://www.fftw.org/>.
- [4] SPIRAL: Software/hardware generation for DSP algorithms. <http://www.spiral.net/>.
- [5] Tensilica: The configurable processor company. <http://www.tensilica.com/html/company.html>.
- [6] International Organization for Standardization. *ISO/IEC 9899-1999, Programming Language: C*, 1999. Technical Committee: JTC 1/SC 22/WG 14.
- [7] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
- [8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.
- [9] Larry McVoy and Carl Staelin. MOB: Memory Organization Benchmark. <http://www.bitmover.com/lmbench/>.
- [10] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference, January 22–26, 1996. San Diego, CA*, pages 279–294, Berkeley, CA, USA, January 1996.
- [11] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
- [12] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect of benchmark run. Technical Report CSD-93-767, February 1993.
- [13] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *USENIX 1998 Annual Technical Conference, January 15–18, 1998. New Orleans, Louisiana*, pages 155–166, Berkeley, CA, USA, June 1998.
- [14] R. Clint Whaley, Antoine Petit, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).
- [15] Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
- [16] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. Technical Report TR2004-1970, November 2004.